发指南（十）——JNI局部引用、全局引用和弱全局引用

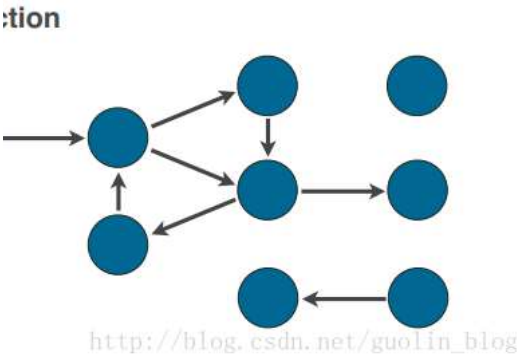
-04-04 16:28:49 15431 收藏 17 版权

K JNI/NDK开发指南 文章标签： 局部引用 全局引用 弱全局引用 引用比较 引用生命周期

blog.csdn.net/xyang81/article/details/44657385

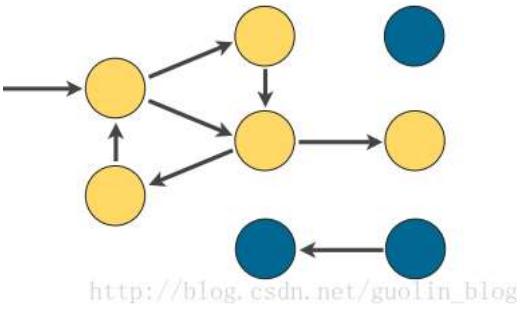
，详细介绍了在编写本地代码时三种引用的使用场景和注意事项。可能看起来有点枯燥，但引用是在点，如果使用不当，容易使程序造成内存溢出，程序崩溃等现象。所以讲得比较细，有些地方看起来可《Android JNI局部引用表溢出：local reference table overflow (max=512)》这篇文章是一个JNI引用使最终导致程序崩溃的例子。建议看完这篇文章之后，再去看。

道，在编码的过程当中，内存管理这一块完全是透明的。new一个类的实例时，只知道创建完这个类的实创的一个引用，然后就可以拿着这个引用访问它的所有数据成员了（属性、方法）。完全不用管JVM内部所创建的对象来申请内存，也不用管对象使用完之后内存是怎么释放的，只需知道有一个垃圾回器在帮忙。有经验的朋友也许知道启动一个Java程序，如果没有手动创建其它线程，默认会有两个线程在跑，一就是GC线程（负责将一些不再使用的对象回收）。如果你曾经是做Java的然后转去做C++，会感觉很“蛋对象，使用完了还要做一次delete操作，malloc一次同样也要调用free来释放相应的内存，否则你的程序且在C/C++中内存还分栈空间和堆空间，其中局部变量、函数形参变量、for中定义的临时变量所分配的内则（而且还要注意大小的限制），用new和malloc申请的内存都存放在堆空间。。。但C/C++里的内存管些只是最基础的内存管理常识。做Java的童鞋听到这些肯定会偷乐了，咱写Java的时候这些都不用管，了。手动管理内存虽然麻烦，而且需要特别细心，一不小心就有可能造成内存泄露和野指针访问等程序利弊，手动申请和释放内存对程序的掌握比较灵活，不会受到平台的限制。比如我们写Android程序的ivk虚拟机的限制，从最初版本的16~24M，到后来的32M到64M，可能随着以后移动设备物理内存的不版本内存限制可能也会随着提高。但在C/C++这层，就完全不受虚拟机的限制了。比如要在Android中要刚好这张图片的大小超过了Dalivk虚拟机对每个应用的内存大小限制，Java此时就显得无能为力了，但碟了，malloc(1024*1024*50)，要多少内存，您说个数。。。C/C++程序员得意的说道～～Java不是说吗，所以除了基本数据类型外，其它任何类型所创建的对象，JVM所申请的内存都存在堆空间。上面提不再使用的对象，它的全称是Garbage Collection，也就是所谓的垃圾回收。JVM会在适当的时机触发操作，就会将一些不再使用的对象进行回收。那么哪些对象会被认为是不再使用，并且可以被回收的呢？（注：图摘自博主郭霖的《Android最佳性能实践(二)——分析内存的使用情况》）



圈就代表一个内存当中的对象，而圆圈之间的箭头就是它们的引用关系。这些对象有些是处于活动状态被使用了。那么GC操作会从一个叫作Roots的对象开始检查，所有它可以访问到的对象就说明还在使用当它的对象就表示已经不再被使用了，如下图所示：

tion



的对象都处于活动状态，仍然会被系统继续保留，而蓝色的对象就会在GC操作当中被系统回收掉了，的简单流程。

点多哈，下面进入正题。通过上面的讨论，大家都知道，如果一个Java对象没有被其它成员变量或静态有可能会被GC回收掉。所以我们在编写本地代码时，要注意从JVM中获取到的引用在使用时被GC回收不能直接通过引用操作JVM内部的数据结构，要进行这些操作必须调用相应的JNI接口来间接操作所引了和Java相对应的引用类型，供本地代码配合JNI接口间接操作JVM内部的数据内容使用。如：jobject、jintArray等。因为我们只通过JNI接口操作JNI提供的引用类型数据结构，而且每个JVM都实现了JNI规范不必担心特定JVM中对象的存储方式和内部数据结构等信息，我们只需要学习JNI中三种不同的引用即

虚拟机中的这个特点，在Java中创建的对象、定义的变量和方法，内部对象的数据结构是怎么定义道。如果我们在C/C++中想要访问Java中对象的属性和方法时，是不能够直接操作JVM内部Java对象在C/C++中正确的访问Java的数据结构，JVM就必须有一套规则来约束C/C++与Java互相访问的机规范，JNI规范定义了一系列接口，任何实现了这套JNI接口的Java虚拟机，C/C++就可以通过调用这一问Java中的数据结构。比如前面文章中学习到的常用JNI接口有：GetStringUTFChars（从Java虚拟机、ReleaseStringUTFChars（释放从JVM中获取字符串所分配的内存空间）、NewStringUTF、etFieldID、GetMethodID、FindClass等。

及区别

三种引用：局部引用（Local Reference）、全局引用（Global Reference）、弱全局引用（Weak别如下：

LocalRef和各种JNI接口创建（FindClass、NewObject、GetObjectClass和NewCharArray等）。会阻止不在本地函数中跨函数使用，不能跨线前使用。函数返回后局部引用所引用的对象会被JVM自动释放，释放。 `(*env)->DeleteLocalRef(env,local_ref)`

```
.ng = (*env)->FindClass(env, "java/lang/String");
'Arr = (*env)->NewCharArray(env, len);
j = (*env)->NewObject(env, cls_string, cid_string, elemArray);
j_local_ref = (*env)->NewLocalRef(env,str_obj);   // 通过NewLocalRef函数创建
```

GlobalRef基于局部引用创建，会阻GC回收所引用的对象。可以跨方法、跨线程使用。JVM不会自动释balRef手动释放 `(*env)->DeleteGlobalRef(env,g_cls_string);`

```
;_cls_string;
INIEnv* env, jobject obj) {
_string = (*env)->FindClass(env, "java/lang/String");
ıg = (*env)->NewGlobalRef(env,cls_string);
```

ewWeakGlobalRef基于局部引用或全局引用创建，不会阻止GC回收所引用的对象，可以跨方法、跨线释放，在JVM认为应该回收它的时候（比如内存紧张的时候）进行回收而被释放。或调用手动释放。 `(*env)->DeleteWeakGlobalRef(env,g_cls_string)`

```
;_cls_string;
INIEnv* env, jobject obj) {
_string = (*env)->FindClass(env, "java/lang/String");
```

```
ıg = (*env)->NewWeakGlobalRef(env,cls_string);
```

通常是在函数中创建并使用。会阻止GC回收所引用的对象。比如，调用NewObject接口创建一个新的这个对象的局部引用。局部引用只有在创建它的本地方法返回前有效，**本地方法返回到*Java*层之后，如局部引用使用的话**，局部引用就会被JVM自动释放。你可能会为了提高程序的性能，在函数中将局部引存起来，供下次调用时使用。这种方式是错误的，因为函数返回后局部引很可能马上就会被释放掉，静态被释放后的内存地址，成了一个野针对，下次再使用的时候就会造成非法地址的访问，使程序崩溃。请看缓存了String的Class引用：

```
*/
.ng JNICALL Java_com_study_jnilearn_AccessCache_newString
jobject obj, jcharArray j_char_arr, jint len)

 elemArray;
's = NULL;
;tr = NULL;
ıss cls_string = NULL;
:hodID cid_string = NULL;
误的引用缓存
'ing == NULL) {
'ing = (*env)->FindClass(env, "java/lang/String");
;_string == NULL) {
:urn NULL;


ng的构造方法ID
'ing == NULL) {
'ing = (*env)->GetMethodID(env, cls_string, "<init>", "([C)V");
l_string == NULL) {
:urn NULL;


弋码.......
= (*env)->NewCharArray(env, len);

ənv)->NewObject(env, cls_string, cid_string, elemArray);
引用
.eteLocalRef(env, elemArray);
:r;
```

了和我们讨论无关的代码。因为FindClass返回一个对java.lang.String对象的局部引用，上面代码中缓存

假设一个本地方法C.f调用了newString：

```
.ng JNICALL
ıv *env, jobject this)

:r = ...;

ıString(c_str);
```

learn_AccessCache_newString 下面简称newString

释放在这个方法执行期间创建的所有局部引用，也包含对String的Class引用cls_string。当再次调用g所指向引用的内存空间已经被释放，成为了一个野指针，再访问这个指针的引用时，会导致因非法的内

第一次调是OK的

```
' 第二次调用时，访问的是一个无效的引用.
```

中方式，一个是本地方法执行完毕后JVM自动释放，另外一个是自己调用DeleteLocalRef手动释放。既然自动释放所有局部引用，为什么还需要手动释放呢？大部分情况下，我们在实现一个本地方法时不必担心函数被调用完成后，JVM 会自动释放函数中创建的所有局部引用。尽管如此，以下几种情况下，为了避手动释放局部引用：
引用都存储在一个局部引用表中，如果这个表超过了最大容量限制，就会造成局部引用表溢出，使程序崩的JNI局部引用表最大数量是512个。当我们在实现一个本地方法时，可能需要创建大量的局部引用，如可能导致JNI局部引用表的溢出，所以，在不需要局部引用时就立即调用DeleteLocalRef手动删除。比本地代码遍历一个特别大的字符串数组，每遍历一个元素，都会创建一个局部引用，当对使用完这个元素马上手动释放它。

```
: len; i++) {
;tr = (*env)->GetObjectArrayElement(env, arr, i);
用jstr */
eleteLocalRef(env, jstr); // 使用完成之后马上释放
```

时，工具函数在程序当中是公用的，被谁调用你是不知道的。上面newString这个函数演示了怎么样在工用后，调用DeleteLocalRef删除。不这样做的话，每次调用newString之后，都会遗留两个引用占用空间，cls_string不用static缓存的情况下）。
会返回。比如一个接收消息的函数，里面有一个死循环，用于等待别人发送消息过来 `while(true) { if 。。。} else { 等待新的消息。。。}}` 。如果在消息循环当中创建的引用你不显示删除，很快将会造成

用的对象被GC回收。比如你写的一个本地函数中刚开始需要访问一个大对象，因此一开始就创建了一个在函数返回前会有一个大量的非常复杂的计算过程，而在这个计算过程当中是不需要前面创建的那个大对计算的过程当中，如果这个大对象的引用还没有被释放的话，会阻止GC回收这个对象，内存一直占用所以这种情况下，在进行复杂计算之前就应该把引用给释放了，以免不必要的资源浪费。

```
本地方法实现 */
 JNICALL Java_pkg_Cls_func(JNIEnv *env, jobject this)

            /* lref引用的是一个大的Java对象 */
            /* 在这里已经处理完业务逻辑后，这个对象已经使用完了 */
eteLocalRef(env, lref); /* 及时删除这个对这个大对象的引用，GC就可以对它回收，并释放相应的资源*/
ıtation();   /* 在里有个比较耗时的计算过程 */
            /* 计算完成之后，函数返回之前所有引用都已经释放 */
```

管理局部引用的生命周期。这些函数包括：EnsureLocalCapacity、NewLocalRef、PushLocalFrame、eLocalRef。JNI规范指出，任何实现JNI规范的JVM，必须确保每个本地函数至少可以创建**16个局部引用**支持创建16个局部引用）。实际经验表明，这个数量已经满足大多数不需要和JVM中内部对象有太多交需要创建更多的引用，可以通过调用EnsureLocalCapacity函数，确保在当前线程中创建指定数量的局部返回0，否则创建失败，并抛出OutOfMemoryError异常。EnsureLocalCapacity这个函数是1.2以上版本才
在编译的时候，如果申请创建的局部引用超过了本地引用的最大容量，在运行时JVM会调用FatalError在开发过程当中，可以为JVM添加-verbose:jni参数，在编译的时如果发现本地代码在试图申请过多的引用我们要注意。在下面的代码中，遍历数组时会获取每个元素的引用，使用完了之后不手动删除，不考虑可以为这种创建大量的局部引用提供足够的空间。由于没有及时删除局部引用，因此在函数执行期间，会

```
，确保函数能创建len个局部引用*/
ıreLocalCapacity(env,len) != 0) {
en个局部引用的内存空间失败 OutOfMemoryError*/
```

```
en; i++) {
:r = (*env)->GetObjectArrayElement(env, arr, i);
jstr字符串
```

```
除在for中临时创建的局部引用*/
```

Capacity函数可以扩充指定容量的局部引用数量外，我们也可以利用Push/PopLocalFrame函数对创建作用。例如，我们把上面那段处理字符串数组的代码用Push/PopLocalFrame函数对重写：

```
... /*最大局部引用数量*/
 len; i++) {
->PushLocalFrame(env, N_REFS) != 0) {
内存溢出*/

tr = (*env)->GetObjectArrayElement(env, arr, i);
用jstr */
opLocalFrame(env, NULL);
```

函数中需要用到的局部引用创建了一个引用堆栈，（如果之前调用PushLocalFrame已经创建了Frame，仍然是有效的)每遍历一次调用 (*env)->GetObjectArrayElement(env, arr, i); 返回一个局部引用时，入当前局部引用栈中。而PopLocalFrame负责销毁栈中所有的引用。这样一来，Push/PopLocalFrame函生命周期更方便的管理，而不需要时刻关注获取一个引用后，再调用DeleteLocalRef来释放引用。在上面str的过程当中又创建了局部引用，则PopLocalFrame执行时，这些局部引用将全都会被销毁。在调用前frame中的所有引用前，如果第二个参数result不为空，会由result生成一个新的局部引用，再把这个新二一个frame中。请看下面的示例：

```
L *PopLocalFrame)(JNIEnv *env, jobject result);

jstr;
 len; i++) {
->PushLocalFrame(env, N_REFS) != 0) {
内存溢出*/

tr = (*env)->GetObjectArrayElement(env, arr, i);
用jstr */
) {
jstr = jstr;

 = (*env)->PopLocalFrame(env, other_jstr);  // 销毁局部引用栈前返回指定的引用
```

局部引用不能跨线程使用，只在创建它的线程有效。不要试图在一个线程中创建局部引用并存储到全局个线程中使用。

跨线程使用，直到它被手动释放才会失效。同局部引用一样，也会阻止它所引用的对象被GC回收。与局是，只能通过NewGlobalRef函数创建。下面这个版本的newString演示怎么样使用一个全局引用：

```
ng JNICALL Java_com_study_jnilearn_AccessCache_newString
jobject obj, jcharArray j_char_arr, jint len)


r = NULL;
ss cls_string = NULL;
ing == NULL) {
 local_cls_string = (*env)->FindClass(env, "java/lang/String");
_string == NULL) {
urn NULL;


va.lang.String类的Class引用缓存到全局引用当中
ing = (*env)->NewGlobalRef(env, local_cls_string);

局部引用
->DeleteLocalRef(env, local_cls_string);

验证全局引用是否创建成功
```

```
s_string == NULL) {
urn NULL;
```

```
';
```

alWeakRef 创建，使用 DeleteGlobalWeakRef 释放。下面简称弱引用。与全局引用类似，弱引用可以跨方
弱引用很重要不同的一点是，弱引用不会阻止GC回收它引用的对象。在newString这个函数中，我们也可
ng的Class引用，因为java.lang.String这个类是系统类，永远不会被GC回收。当本地代码中缓存的引用
所指向的对象时，弱引用就是一个最好的选择。假设，一个本地方法mypkg.MyCls.f需要缓存一个指向类
如果在弱引用中缓存的话，仍然允许mypkg.MyCls2这个类被unload，因为弱引用不会阻止GC回收所引
代码段：

```
JNICALL
.s_f(JNIEnv *env, jobject self)

ass myCls2 = NULL;
 == NULL)

 myCls2Local = (*env)->FindClass(env, "mypkg/MyCls2");
Cls2Local == NULL)

urn; /* 没有找到mypkg/MyCls2这个类 */

 = NewWeakGlobalRef(env, myCls2Local);
Cls2 == NULL)

urn; /* 内存溢出 */


myCls2的引用 */
```

2有相同的生命周期（例如，他们可能被相同的类加载器加载），因为弱引用的存在，我们不必担心
代码在被使用时，MyCls2这个类出现先被unload，后来又会preload的情况。当然，如果真的发生这种情
比时的生命周期不同），我们在使用弱引用时，必须先检查缓存过的弱引用是指向活动的类对象，还是指
ad的类对象。下面马上告诉你怎样检查弱引用是否活动，即引用的比较。

全局、局部还是弱全局引用），我们只需要调用IsSameObject来判断它们两个是否指向相同的对象。例
ject(env, obj1, obj2)，如果obj1和obj2指向相同的对象，则返回JNI_TRUE（或者1），否则返回
有一个特殊的引用需要注意：NULL，JNI中的NULL引用指向JVM中的null对象。如果obj是一个局部或
>IsSameObject(env, obj, NULL) 或者 obj == NULL 来判断obj是否指向一个null对象即可。但需要注意的
弱全局引用与NULL比较时，返回值的意义是不同于局部引用和全局引用的：

```
bj_ref = (*env)->NewObject(env, xxx_cls,xxx_mid);
ef = (*env)->NewWeakGlobalRef(env, local_ref);
处理
l = (*env)->IsSameObject(env, g_obj_ref, NULL);
```

周用中，**如果g_obj_ref指向的引用已经被回收，会返回JNI_TRUE，如果wobj仍然指向一个活动对象，**

时，除了它所指向的JVM中对象的引用需要占用一定的内存空间外，引用本身也会消耗掉一个数量的
优秀的程序员，我们应该对程序在一个给定的时间段内使用的引用数量要十分小心。短时间内创建大
的引用很可能就会导致内存溢出。

不再需要一个全局引用时，应该马上调用DeleteGlobalRef来释放它。如果不手动调用这个函数，即使这
M也不会回收这个全局引用所指向的对象。
地代码不再需要一个弱全局引用时，也应该调用DeleteWeakGlobalRef来释放它，如果不手动调用这个函
JVM仍会回收弱引用所指向的对象，但弱引用本身在引用表中所占的内存永远也不会被回收。

则

一个全面的介绍，下面来总结一下引用的管理规则和使用时的一些注意事项，使用好引用的目的就是为了
引用保持而不能释放，造成内存浪费。所以在开发当中要特别小心！
地代码使用引用时要注意：
引的native函数的本地代码
要当心不要造成全局引用和弱引用的累加，因为本地方法执行完毕后，这两种引用不会被自动释放。
工具函数。例如：方法调用、属性访问和异常处理的工具函数等。
引时，要当心不要在函数的调用轨迹上遗漏任何的局部引用，因为工具函数被调用的场合和次数是不确定
能很有可能造成内存溢出。所以在编写工具函数时，请遵守下面的规则：
型的工具函数被调用时，它决不能造成局部、全局、弱全局引用被回收的累加
类型的工具函数被调用时，它除了返回的引用以外，它决不能造成其它局部、全局、弱引用的累加
了使用缓存技术而创建一些全局引用或者弱全局引用是正常的。如果一个工具函数返回的是一个引用，我
明返回引用的类型，以便于使用者更好的管理它们。下面的代码中，频繁地调用工具函数GetInfoString，
ing返回引用的类型是什么，以便于每次使用完成后调用相应的JNI函数来释放掉它。

```
JE) {
nfoString = GetInfoString(info);
里infoString */
用完成之后，调用DeleteLocalRef、DeleteGlobalRef、DeleteWeakGlobalRef哪一个函数来释放这个引用呢？*/
```

被用来确保一个工具函数返回一个局部引用。我们改造一下newString这个函数，演示一下这个函数的用
把一个被频繁调用的字符串“CommonString”缓存在了全局引用里：

```
.ng JNICALL Java_com_study_jnilearn_AccessCache_newString

ring result;
ncmp函数比较两个Unicode字符串 */
mp("CommonString", chars, len) == 0)

ommonString"这个字符串缓存到全局引用中 */
jstring cachedString = NULL;
chedString == NULL)

先创建"CommonString"这个字符串 */
ring cachedStringLocal = ...;
然后将这个字符串缓存到全局引用中 */
chedString = (*env)->NewGlobalRef(env, cachedStringLocal);

全局引用创建一个局引用返回，也同样会阻止GC回收所引用的这个对象，因为它们指向的是同一个对象
(*env)->NewLocalRef(env, cachedString);


lt;
```

期中，Push/PopLocalFrame是非常方便且安全的。我们可以在本地函数的入口处调用
在出口处调用PopLocalFrame，这样的话，在函数内任何位置创建的局部引用都会被释放。而且，这两
强烈建议使用它们。需要注意的是，如果在函数的入口处调用了PushLocalFrame，记住要在函数所有出
地方）都要调用PopLocalFrame。在下面的代码中，对PushLocalFrame的调用只有一次，但调用
次，当然你也可以使用goto语句来统一处理。

```
v *env, ...)

ult;
->PushLocalFrame(env, 10) < 0)

PushLocalFrame获取10个局部引用失败，不需要调用PopLocalFrame */
NULL;
```

```
.; // 创建局部引用result

前先弹出栈顶的frame */
= (*env)->PopLocalFrame(env, result);
result;

'env)->PopLocalFrame(env, result);
 */
ılt;
```

数PopLocalFrame的第二个参数的用法，局部引用result一开始在PushLocalFrame创建在当前frame里 .ocalFrame中时，PopLocalFrame在弹出当前的frame前，会由result生成一个新的局部引用，再将这个 上一个frame当中。

a数组 和 JNI引用) System.o的博客 1721
数组元素个数（数组长度） size GetArrayLength(JNIEnv *env, jarray array) //返回对象数组元素中的对象 jobject Get…

引用 文韬武略的专栏 5732
方法内的局部变量，会随着方法调用完return后，局部变量也会随着被释放。 所以，不要在本地方法中定义static变…

者获得更高权重 评论

我是雅马哈，中国的骄傲~ 2 年前 回复

好，请教一个问题啊，关于局部引用的生命周期仅是方法内部和线程内部，但在源码Android_os_messageques xceptionObj，这个变量是全局的，但为什么会被当成局部引用来使用。 [code=cpp] NativeMessageQueue::Nativ ıPollEnv(NULL), mPollObj(NULL), mExceptionObj(NULL) { mLooper = Looper::getForThread(); if (mLooper == N / Looper(false); Looper::setForThread(mLooper); } } NativeMessageQueue::~NativeMessageQueue() { } void Na aiseException(JNIEnv* env, const char* msg, jthrowable exceptionObj) { if (exceptionObj) { if (mPollEnv == env) nv->DeleteLocalRef(mExceptionObj); } mExceptionObj = jthrowable(env->NewLocalRef(exceptionObj)); ALOGE eQueue callback: %s", msg); jniLogException(env, ANDROID_LOG_ERROR, LOG_TAG, exceptionObj); } else s", msg); jniLogException(env, ANDROID_LOG_ERROR 3 年前 回复

明白，关于CommonString中这个例子，最后该释放什么，不该释放什么？ 5 年前 回复

登录 查看 11 条热评

han 3-12
它允许Java 代码和其他语言写的代码进行交互。JNI 一开始是为了本地已编译语言,尤其是C 和C++而设计的,但是它…

g的专栏 4-3
简单的JNI的调用的过程。 JAVA以其跨平台的特性深受人们喜爱,而又正由于它的跨平台的目的,使得它和本地机器的…

引用 shuimuniao的专栏 926
引用：局部引用(local reference)、全局引用(global reference)以及弱全局引用(weak global reference)。三种类型的…

用，弱引用 mashaoshuai12的专栏 169
要是 局部引用和全局引用 // 作用：在JNI中告知虚拟机何时回收一个JNI变量 // 局部引用，通过DeleteLocalRef手动…

用_李国菁LGJ的博客 3-14
用对象: 1.访问一个很大的java对象,使用完之后,还要进行复杂的耗时操作 2.创建了大量的局部引用,占用了太多的内存…

)——Java与JNI互相调用_xingfe... 4-7
ava的反射机制,需要首先找到类、再找到某个方法或字段,再进行调用。 这里涉及JNIEnv的几个方法: //根据全限定名…

局部和全局引用 Jerry Lin 3427
不透明的引用。native代码从不会直接检查一个不透明的引用指针的上下文，而是通过使用JNI函数来访问由不透明…

Sailingthink的专栏 1059

NI局

点赞18 评论11 分享 收藏17 打赏 举报 关注 一键三连

作 厚积薄发，持之以恒 393
式传递给原生代码，因此引用类型不能直接使用和修改。但JNI提供了一些操作应用类型的API 本文主要介绍 字符串 …

[局部引用、全局引用、弱全局引用] [NewObject等、NewGlobalRef、New… Hendy_Raw的博客 1750
¥众多接口返回的引用 下例以FindClass返回的引用，存储于全局变量中，或静态变量中。 第一次运行，创建，正常…

引用、弱全局引用 AND 缓存jfieldID和jmethodID的两种方法 as371418912的专栏 1298
引、弱全局引用 AND 缓存jfieldID和jmethodID的两种方法 本文

引用 pecuyu 3518
四种引用类型，分别是：强引用、软引用、弱引用以及虚引用。对于一个Java对象来说，当被强引用所引用时，只…

全局引用和弱全局引用 u013187531的博客 153
1190000005744548

和弱全局引用 Zh_hao的博客 232
mp;nbsp; 版权声明：本文为博主原创文章，未经博主允许不得转载。 …

3-JNI的局部和全局引用 Android研发专栏 1881
返回局部引用。局部引用不能在后续的调用中被缓存及重用，主要是因为它们的使用期限仅限于原生方法，一旦原生…

NI中的全局引用/局部引用/弱全局引用、缓存jfieldID和jmethodID的两种方式... 54
引用/弱全局引用、缓存jfieldID和jmethodID的两种方式，并编写两种缓存方式的示例代码。 1.从Java虚拟机创建的对…

D全局引用 Tonyfield的专栏 3226
/docs/books/jni/html/refs.html JNI给出实例和数组类型（如jobject，jclass，jstring，jarray）作为不透明的引用。N…

二) JNI局部引用、全局引用和弱全局引用 weixin_34352005的博客 102
ython工程师标准>>> ...

——JNI局部引用、全局引用和弱全局引用 weixin_33834628的博客 187
ython工程师标准>>> ...

1371
NI 中如何控制 Java 对象的声明周期，当时没答出来，其实我项目中就有使用的地方，可能还不是那么熟吧，所以下…

NI局部引用和全局引用 Songlcy 1236
的介绍了JNI的字段和方法，想必大家都对JNI与Jvm交互有了更深刻的认识。Android NDK（五）：字段和方法 本篇…

爬行的菜鸟的博客 1683
问题现在整理一下 环境：Android stdio 3.0 工具：cmake 材料：libnative.so 晚点再写一篇专门介绍的，现在只介绍…

xyang0917 专家
码龄13年 暂无认证

103	1万+	3311	195万+	
原创	周排名	总排名	访问	等级
1万+	2222	817	582	743
积分	粉丝	获赞	评论	收藏

私信 关注

搜博主文章

热门文章

深入分析Java ClassLoader原理 150247

Supervisor安装与配置（Linux/Unix进程管理工具） 149084

Keepalived安装与配置 84229

Maven安装与环境配置（Windows） 79427

分类专栏

招聘

JNI/NDK开发指南 14篇

C 10篇

C++ 2篇

Android 13篇

JavaSE 26篇

最新评论

Keepalived+Nginx实现高可用（HA）
lizepengg: 多谢keepalived.conf文件

MySQL5.7安装与配置（YUM）
Tisfy: 真棒！就像：富贵必从勤苦得，男儿须读五车书。

Supervisor安装与配置（Linux/Unix进程…
weixin_46364516: echo_supervisord_conf 这个在哪里?

Supervisor安装与配置（Linux/Unix进程…
weixin_46364516: 第2步没有生三个执行程序

深入分析Java ClassLoader原理
m0_50126769: 你牛逼 写一个让我们瞧!

最新文章

【阿里巴巴-高德-汽车事业部】【内推】Java技术专家、前端技术专家、C++技术专家（长期招聘）

分布式服务管理框架-Zookeeper节点ACL

分布式服务管理框架-Zookeeper客户端zkCli.sh使用详解

2019年 1篇 2016年 39篇
2015年 11篇 2014年 18篇
2013年 11篇 2012年 38篇
2011年 2篇

目录